Atomic Use Case as a Concept to Support the MDE Approach to Web Application Development

Kinh Nguyen
Computer Science & Computer Engineering Department
La Trobe University,Australia
kinh.nguyen@latrobe.edu.au
Tharam Dillon
Faculty of Information Technology
University of Technology Sydney,Australia
tharam@it.uts.edu.au

Abstract

While use case is a popular technique for capturing requirements, the process of proceeding from use cases to subsequent modeling activities is to a large extent still unclear. In this paper, we propose the concept of atomic use case to formally model the functional requirements in support of the MDE approach to web application development. In particular we demonstrate how to construct a precise model for the business logic layer and to establish clear relationships between business logic model to other models such as the domain model, user interface model, navigation model, and the business process model. We also explore how the atomic use case concept can be incorporated into UWE (UML Web Engineering) Methodology.

1. Introduction

In this paper, we propose the concept of atomic use case as a fundamental concept to support the MDE (Model-Driven Engineering) approach to web-based application development. A web application, in particular a web information system, can be seen as having the following major components: (a) web-based components, (b) business logic and data source components, (c) other components such as web-services, portlets, agents, metadata, data mining, etc. We will be largely concerned with the first two kinds of components.

The concept of atomic use [7] is proposed as a solution to what we regard as a long-standing problem of information system, be it relational or object-oriented – namely the problem of analyzing and modeling the behavior of information systems.

One advantage of relational database technology, it is often claimed, is the separation of the static aspect and the dynamic aspect. The modeling of the static aspect (the structure of the database) has been handled in practice reasonably well. In contrast, the dynamic aspect (the operations of the database system,) is handled quite poorly. In the 70s, IFIP (The International Federation on Information Processing) recognized it as a key problem and organized a number conferences, inviting researchers to propose solutions to this problem. A number of proposals were made but none of them were adequate to the task, especially as a practical solution. Consequently, the problem gradually slipped into the phase of being a forgotten problem. Interest in the problem was briefly revived in the late 80s with the emergence of Z notation. Before long, it was realized that Z notation could not deliver the expected results, and the the problem slipped back into the state of a forgotten problem. In the mean time, the industry handles the problem in an informal (hence ambiguous) and ad hoc manner, which incurs huge hidden costs (poor understanding of functional requirements, poor specification for programmers, etc.).

As for object-oriented information systems, first of all, we can observe that UML without OCL would be too imprecise to describe behavior of information systems. An activity diagram can represent at a number of tasks and the flows between them; but it lacks the facilities to describe the tasks in detailed and precise terms. The same is true of sequence diagrams, which can show the sequence of messages in a collaboration, but are not good at describing the detailed effects of such messages. Though there are situations in which the sequence of messages alone can be very helpful (e.g. to explain how enterprise Java beans work, that is, how a client, through a number of stubs, can communicate with the beans on the server), it is not the case for information systems. For information systems, it is the effects of the messages that really tell us what really is going

on. In practice, it is more often than not that after drawing pages and pages of sequence diagrams, when it comes to implementation, we have to ask a whole host of questions all over again. The gap between sequence diagrams and the implementation code is far too big in most cases. In the language of MDE, the model does not have sufficient information for the subsequent transformation act. To strengthen the UML models, OCL has been introduced. Since its inception, many important features have been added, and OCL now appears to have adequate expressive power for specifying complex applications. Given that is the case for OCL, an issue needs to be be seriously considered: how are we to use it? For example, what would be the units of behavior that we are going to use it for? And how are we going to identify these units in practice?

We propose atomic use case as an answer to the problem of precise behavior modeling. In this paper, we will introduce the concept and demonstrate how we can use it to construct a precise model for the business logic layer, and from there, to establish the clear relationships between business logic model and the user interface (of any kind). Finally, we explore how the atomic use case concept can be incorporated into UWE (UML Web Engineering) Methodology and briefly describe how we may apply this concept to the Travel Agency case study.

2. Introductory Example

We will take an example to show what atomic use cases are and the role they play through out the life-cycle. Because we wish to show the role this concept can play in the whole life-cycle, we will need to choose a very simple example.

Problem Statement Consider an application in which we are required to maintain information about a set of employees. Each employee has a unique ID, a name, and a phone number. The ID and name of an employee cannot be changed. System operations, or use cases, include: (1) Add an employee, (2) Delete an employee, (3) Change the phone number of an employee, (4) Retrieve employees by ID, by name, or by phone number.

2.1. Identify and Specify Atomic Use Cases

We can go through the use cases one by one, identify the atomic use cases, and formally specify them. Through this process, as will be shown, we end up with a complete model that captures the full functionality of the application.

Consider the operation (use case) to "Add an Employee." A use case description for this operation can be as follows:

"To add a new employee, first, the user enters the id of the new employee. The system checks to determine if the id is new. If it isn't, an error message will be displayed and the operation is terminated. Otherwise, the user enters next the name and the phone number.The system then creates a new employee with the input data and saves it."

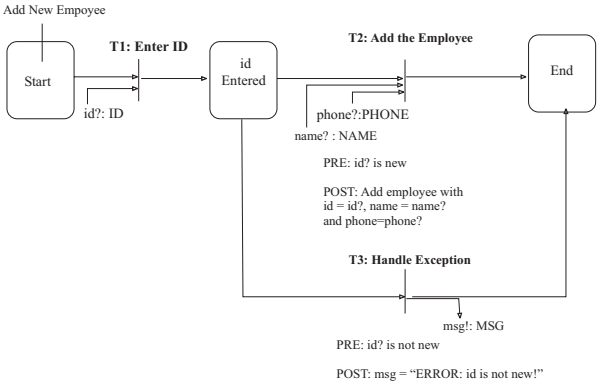


Figure 1. A Petri Net Representation for "Add an Employee" Use Case.

The use case and the various path it can take can be represented by a high-level Petri net shown in Figure 1 (What we present here is a simple version of what we call the "obligation net" [8], a high-level Petri in which each transition has pre- and postconditions to precisely express the obligations the system has to fulfill.)

We now make a shift in the way we view the system's behavior. Instead of thinking in terms of "do this then do that" or "if this, then do that" (the procedural view), we consider the operation as one whole unit and ask "For this use case, in how many different ways can the system makes its response?" (the declarative view). For the current use case, the system can respond in two ways, corresponding to two paths in the system obligation net.

The first path consists of transitions T1 and T2. Following this path, the system makes a *positive* response: it adds a new employee to the information base. The second path consists of transitions T1 and T3. In this case, the system makes a *do-nothing* response: it simply leaves the system in its prior state due to the non-fulfillment of the precondition for adding an employee. We refer to the response described by the first path as an *atomic use case*, and the one described by the second path as an *exception use case*. A general definition can be given as follows:

Definition: *An atomic use case is conceived as an* indivisible *response by the system that either (1) effects a change of the system's state, which takes the system from a consistent state to a consistent state, to reflect an event taking place in the application domain, or (2) performs a query that is of interest to the user in its own right.*

Having identified the atomic use case, and in keeping with the "atomic" viewpoint, we can specify it informally

in terms of the input, output, pre- and postconditions as follows:

– Input: id?, name?, phone?

– Output: None

– Pre: id? is new

– Post: Create a new employee with id?, name?, phone? and add the employee set

We can now seek to specify the atomic use case formally. As expected, the specification always follows a consistent format which consists of input, output, pre- and postconditions, though some of these elements may be absent in a particular case. The "Add Employee" atomic use case can be specified as follows:

AddNewEmployee

input

$id? : ID$

$name? : NAME$

$phone? : PHONE$

pre

$id? \notin \{e : allEmployees \bullet p.id\}$

post

$\exists\, newEmployee : allEmployees' \bullet$

$newEmployee \notin allEmployees$

$newEmployee.id = id?$

$newEmployee.name = name?$

$newEmployee.phone = phone?$

$allEmployees' = allEmployees \cup \{newEmployee\}$

Two points should be made from this example. First, in order to precisely specify the use case, we need to use some formal notation. In the above example, we have used Object-Z [3]. We could use OCL instead. Both use the same mathematical concepts and we can easily translate from one into the other (except for some advanced operators of Object-Z). Object-Z is more concise and we use it here to save space.

Second, we have assumed the existence of two classes (only the static features are required at this stage). One is the *Employee* class, which has three attributes as shown below:

Employee

$id : ID$

$name : NAME$

$phone : PHONE$

...

The other is a system class. It represents the system from the functional view point. It maintains a set of employees and provides method to manage that set of employees. We will call the class *EmployeeApp*:

EmployeeApp

$allEmployees : \mathbb{P}\, Employee$

$\forall e1, e2 \bullet e1 \neq e2 \Leftrightarrow e1.id \neq e2.id$

...

In this example, the *Employee* class is actually the whole of the domain model. In general, to formally specify the atomic use case, we need (a) the domain model or part of the domain model relevant to the use case, and (b) the system class. One crucial aspect of the system class is that through the attributes of the system object we can get to all the domain objects of the system.

By going through the use cases one by one, and identifying and formally specifying the atomic use case for each of them, we would obtain a collection of atomic use case specifications, which constitutes *a complete functional requirements model* of the application.

2.2. Derive Methods for Domain and System Classes to Obtain a Complete Business Logic Model

Once an atomic use case is formally specified, we can deduce the methods required of the domain classes and the system class to support that use case.

Each atomic use case will become a method of the system class. For the "Add a New Employee" use case, we require the method shown below in the system class:

EmployeeApp

AddEmployee

$id? : ID$

$name? : NAME$

$phone? : PHONE$

$id? \notin \{e : allEmployees \bullet e.id\}$

$\exists\, e : Employee \bullet$

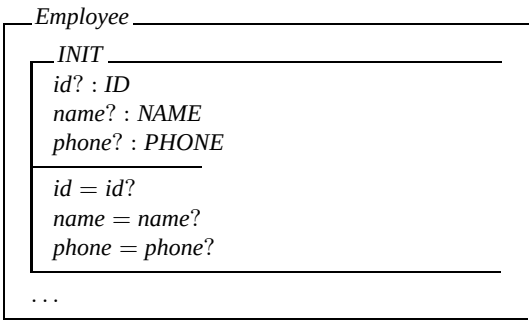
$e \notin allEmployees$

$e.INIT(id?/id?, name?/name?, phone?/phone?)$

$allEmployees' = allEmployee \cup \{e\}$

...

From the postcondition of that method of the system class, it is clear that we need to be able to create new *Employee* instances. Thus, we have identified a method (constructor) required of the domain class to support the use case. The method is expressed in Object-Z as an *INIT* schema as shown below:



By repeating this activity for all atomic use cases, we would get a *complete business logic layer model* as far as the functional requirements are concerned. In general, this model (which strictly speaking is not a design model) consists of a system class and all the domain classes relevant to the particular application.

2.3. Implement and Test the Business Logic Model

Having derived the methods required of the system and domain classes, we can implement or prototype them. An implementation for the "Add New Employee" use case is shown below. We need one method for the system class:

```
public void addNewEmployee( String id,
String name, String phone) throws Exception
{
  // compute the precondition.
  // Assume that collectIds() returns the set of
  // ids of the set of employees
  boolean pre =
      allEmployees.collectIds().contains(id);

  // if precondition is not satisfied,
  // abort the operation
  if (! pre)
  {   throw new Exception
          ( "The ID already exists!" );
  }

  // otherwise, create the new employee and add
  // it to the information base
  Employee newEmployee =
       new Employee(id, name, phone);
  allEmployees.add( newEmployee );
}
```

and one method for the domain class *Employee*:

```
public Employee(String id, String name,
String phone)
{
    this.id = id;
    this.name = name;
    this.phone = phone;
}
```

Note that we have all the details from the business logic model we need to do that. No further discovery activities are needed.

Testing Once the two methods above are available, we can test them with a testing script such as the one shown below. The variable name *theSystem* is to emphasize the fact that an instance of *EmployeeApp* represents the complete system in this example.

```
// create a system object
EmployeeApp theSystem = new EmployeeApp();

// add first employee and display the system's
// state
theSystem.addNewEmployee("E10","Smith","1234");
System.out.println( theSystem.toString() );

// add second employee and display the
// system's state
theSystem.addNewEmployee("E20","Adams","2345");
System.out.println( theSystem.toString() );

// try to add employee with an existing id
// and observe that the system's state remains
// the same
theSystem.addNewEmployee("E10","Clarke","3456");
System.out.println( theSystem.toString() );
```

The testing script contains several test cases (scenarios of the current use case), which are based on the precondition specified in the atomic use case. For each test, the state of the system object is displayed to verify that the implementation satisfies the postcondition specified in the atomic use case.

The tests reveal two important consequences of the implementation of the atomic use case.

- First, we can perform the required operation, i.e. adding employees.
- Second, and just as important, the system can protect itself from invalid requests and preserves the integrity of its state.

The ability to make appropriate responses to both valid and invalid requests is, of course, exactly what we should be looking for. Notice that we can achieve all of these by considering *only* the atomic use cases.

The Functional Core Once we proceed to implement the rest of the operations specified in the business logic model, the two classes that we obtain constitute an executable component that allows us to perform all the required operations (add and delete employees, change phone numbers, etc.). We call this the *functional core*. It is a "basic core of the system" that is fully functional in the sense that it can store the relevant information, update the information, and respond to queries in support of the business activities. For this reason, we take the functional core to be the *business logic layer* of the system. That is, in our approach, we take the business logic layer to be precisely the implementation of the identified *atomic* use cases.

Moreover, once the functional core has been fully tested, we can build the desired graphical user interface as a separate layer on top of it. As will be shown, it is possible to build a separate GUI layer that interacts with the functional core only at a small number of well-defined points.

3. Further Clarification on the Concept of Atomic Use Case

In the definition of atomic use case, given earlier, the criterion that an update atomic use case must *"take the system from a consistent state to a consistent state"* is significant and is useful to identify atomic use cases. The following simple example illustrates this point.

Example - Enroll Student Consider the case of enrolling students in subjects. Suppose subjects are classified as *core* or *optional*, and each student must take at least 3 core subjects. Without the condition that a student must take at least 3 core subjects, the act of enrolling a student in a subject is an atomic use case. With that condition in place, that act is no longer an atomic use case: it may cause the information base to be in an inconsistent state. The atomic use case in this case must be "To enroll a student in a set of subjects in one go". More precisely,

– The inputs are a student id and a set of subject codes

– The outputs: NONE (it is clearer not to regard error messages as output; they are implied by the preconditions)

– The preconditions are: (1) the id must exist, (2) each unit must exists, (3) the set contains at least three core unit

– Postcondition: Enroll the student in those subjects.

Similarly, the phrase *"to reflect an event taking place in the application domain"* provides a useful criterion for identifying atomic use case. The following example illustrates this point.

Example - Add Student or Staff Consider an application which deals with students and staff in an academic institution (suppose we maintain some different information about them). "Add a Student" and "Add a Staff" are atomic use cases. In one of our presentations, it has been asked: Should we take "Add a Person" as an atomic use case? The answer is "No". In the application domain, we may have the event of "Having a new student" or "Having a new staff member", but not the event "Having a new person". "Person' is an abstraction with some information left out of student or staff, and the so-called event "Having a New Person" cannot fully describe the situation. Furthermore, it we consider the states of the information base, we can see this clearer: When the information base change from state S to state S', then S' may be S plus information about a new student or a new staff, but *not* simply about a new person.

Finally, when the use case is a query use case it needs to be *"of interest to the user in its own right*." The following example serves to illustrate this point.

Example - Redistribution Parts between Warehouses In [4], Jacobson presents a rather sophisticated screen to show how the user may interact with a system to redistributes parts among various warehouses (we move items from the 'From' warehouse to the 'To' warehouses). There is a drop-down list to select the 'From' warehouse. When a 'From' warehouse is selected by the user, the system respond by listing the rest of the warehouses as potential 'To' warehouses. In making this response, the system would need to perform a query against the information base. Now, it is unlikely that such a query is of interest to a user by itself (in this example, it serves as a small step in determining the potential 'To' warehouse and where in that warehouse we should move items to). If that is the case, the query does not amount to an atomic use case. It is simply a query that supports the user interface.

In our approach, we would extend the functional core to provide the user interface-support queries. The functional core and the extended part together are called the *extended function core*

4. Relationships to Use case Descriptions Graphical User Interfaces

Use cases can be given at different grains of granuality. Three main levels are usually distinguished, and using the terminology of [2],they are: summary goal level (business use cases), user goal level (system use cases), and subfuctions (subfunction use cases). Use case descriptions are usually given in three general formats: the simple unstructured format, the user-system dialog format, the flows of events format. Given a description of a use case, regardless of its format, we can identify the atomic use case associated with it. As shown earlier, one way to do this is to sketch a net like one in Figure 1 and observe how the system responds to various paths (some lead to atomic use cases, some do not). Very often, we can even recognize the atomic use cases directly: they normally correspond to the main flows of the system use cases.

Identifying atomic use cases through use case descriptions is not the only option. In fact, it is more practical to do so through the graphical user interface. In the industry, people are less likely to talk about use cases; they often talk about user interface and how the user interacts with the user interface. We will be talking the 'language of the industry' when we identify the atomic use cases through the user interface sketches (or designs) and the descriptions of how they work.

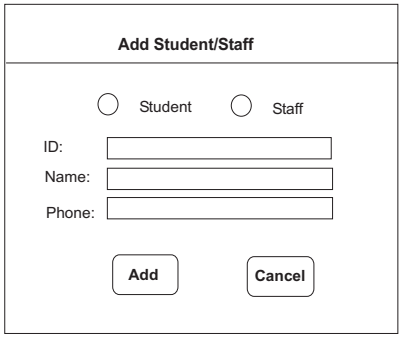


Figure 2. Screen to Add Student or Staff.

Example - Add Staff or Student Consider the screen in Figure 2.

– When the user checks either the 'Student' or the 'Staff' checkbox, no query is made to the information base. The user interface simply 'remembers' that the user has made that choice.

– When we enter id, the system would respond by checking whether the id is new or not, and warn us if it's not new. This action requires the system to make a query to the underlying information base. The purpose of this query is to support the user interface.

– After the user has entered relevant information and press button "ADD", the system will respond by calling the business layer to add a student or a staff member. Thus, out of this screen we have two atomic use cases: one to add a new staff and one to add a new student.

In general, given a screen or a series of screens, web-based or otherwise, many events can be generated by the user's actions. By examining how the system responds to each of these events, in particular how it interacts with the information base, we can identify atomic use cases and all the queries needed to support the operation of the user interface.

5. Transforming Functional Core Model to Business Logic Layer

The implementation of the functional core model, in essence, requires the translation of formal expressions for pre- and postconditions into programming code. To do that, in most cases we only need to handle a small number of common expressions. These expressions are are list in Figure 3, which show the equivalents in Smalltalk. Equivalent Java code segments would be less concise, but definitely standard patterns can easily be established.

In the language of MDE, (a) we have a formal platform independent model of the business logic layer, and (b) we can readily formulate transformation rules to transform the model into a platform-specific component.

$\forall x : X \bullet p(x)$	`X size = (X select: [:p| p(x)]) size`
$\exists x : X \bullet p(x)$	`(X select: [:P| p(x)]) isEmpty not`
$x \in X$; $x \notin X$	`x in: X (or X includes: x )`
$x \notin X$	`x notIn: X (or (X includes: x) not)`
$X \cup \{a\}$	`X add: a`
$X \setminus \{a\}$	`X remove: a`
$X \cup Y$	`X union: Y`
$X \cap Y$	`X intersect: Y`
$X \setminus Y$	`X removeAll: Y`
$\{x : X \mid p(x)\}$	`X select: [ :x | p(x) ]`
$\{x : X \bullet e(x)\}$	`X collect: [ :x | e(x) ]`
$\exists x : X \bullet p(x) \blacktriangleright x.oper$	`x := X detect: [ :x | p(x) ]. x oper`

(Note that `in:aCollection` can be defined as a method in the class `Object` as `aCollection includes: self`)

Figure 3. Equivalent Expressions between Formal Specification and Smalltalk

EJB Implementation We can implement each atomic use case as a session bean, which may access data sources in a distributed manner. In this case, the system class is implemented implicitly as a collection of session beans.

However, in most cases, it is better to put all the atomic use cases in one session bean, one method for each atomic use case. In this case, the system class is implemented explicitly. Conceptually, such an implementation would be much clearer and easier to understand. Practically, it would make it easier for the client programs to locate and use the beans on the server-side: there is only one bean that the client should be aware of.

Note It is not unusual to break up an application into a small number of major components. In such a case, the "system class" is implicitly made up of these components and each atomic use case is to be described in the context of a particular component. Also, we must specify the interactions among the components in terms of the messages they can send to each other.

Relational Implementation It is not uncommon to implement the information base as a relational database and

to have the business logic code written in a non-object-oriented fashion (e.g. using JDBC instead of Hibernate, say). Then how would we proceed from the specification to the implementation?

In this case, we need to convert the object-oriented model into a relational model. That is, we convert the class diagram into a relational schema, and each atomic use case must be expressed as a method that acts on the relational schema. The first task (to obtain the relational schema) is rather straight forward and many mapping rules have been suggested. The second task (to capture the behavior), at the intuitive level is very straight forward. Moreover, we only need to be concerned with a small number of expressions (that are used to express the pre- and postconditions) presented earlier.

Alternatively, we could choose to write the behavioral specification based on the relational model from the outset. In this case, standard Z notation is not quite suitable for practical use (as witnessed by history). We have experimented with an extension of Z, which we call 'RZ', 'R' for 'relational'. It does not have (yet) a formal semantics, though intuitively the meanings of the additional constructs are clear. As a kind of 'formal pseudo code', RZ works really well (quite unambiguously) in expressing the intentions of the modeler. In addition, for the consumption of those who do not want to use mathematical notations, we have added a few features to SQL to form a pseudo code language to express the atomic use cases (i.e. their pre- and post-conditions) in a highly (but not formally) precise manner. We are currently investigating the use of an object-oriented specification language (such as Object-Z, OCL) for specifying pre- and postconditions against a relational schema by representing tuples of a relation as simple objects where all the attributes are publicly accessible for manipulation. The use of Maude [1] could be quite suitable for this task as well.

6. Atomic Use Cases and Events in Web Applications

The relationships between the atomic use cases and the events generated by graphical user interfaces (considered earlier) indicate how atomic use cases can be applied to the analysis, modeling and specification of web applications. An event generated by the user's actions on a web-based interface can be conceived as a contract (to be fulfilled in most cases by the method that handles the event on the server side). This contract consists of

– The input data (they normally come from the HTML forms or cookies or the session object);

– The output (in the contract for a web event, we define the output to be data that we need to pass to the returned web page – see example below);

– The precondition (to specify the conditions that the input data must satisfy);

– The postcondition (to specify actions such as updating of the cookies or session object, performing query or update action against the business logic layer, i.e. invoking atomic use cases);

– The returned page (the page is divided into one or more "blocks", and for each block, we specify the input data that it receives and displays and the events it may generate; for each event, we specify its parameters).

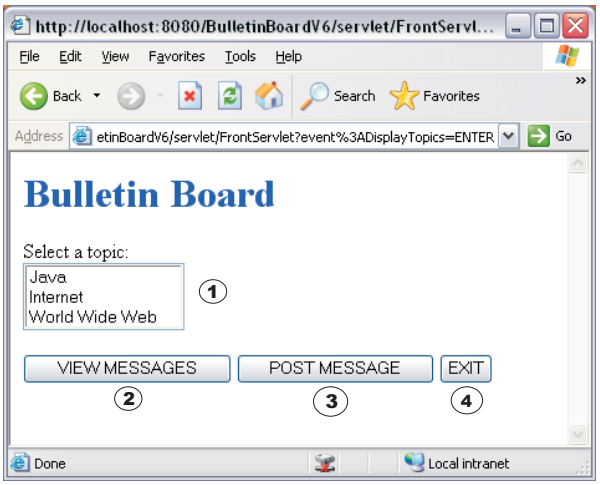
The content of such a contract for a web event can be formally specified. As an example, consider the screen for an online bulletin board shown in Figure 4. Several events can be generated by this screen and the specification for the "View Messages" event, for example, is shown below (the boolean expressions are in OCL):

```
Event ViewMessages
Input:
   // topic? is the name of the selected topic
   Request topic?: String
Output:
   // set of messages to be displayed by the
   // returned web page
   messages!: Set(String)
   // the topic needs to be passed to the
   // returned web page
   topic!
Pre:
   // name of selected topic must exist in the
   // information base
   bulletinBoardFnCore.
       getTopics.name ->includes(topic?)
Post:
   // retrieve the messages and pass the topic
   // along to the returned web page
   messages! =
       bulletinBoardFnCore.
           getMessages(topic?).text
   topic! = topic?
Returned Page:
   Block 1:
       Input:
           message! : Set(String)
           topic!: String
       Next Events:
           DisplayTopics(Session topic: String)
               where topic = topic!
```

Notice that (a) In the Post section, a call is made to the business logic layer; and (b) The returned page contains the Display Topics event whose argument is the topic received by the View Messages event by virtue of two constraints: `topic! = topic?` in the Post section, and `topic = topic!` in the Next Events subsection of the Returned Page section.

7. Exploring the Application of Atomic Use Cases to UWE

UWE (UML-based Web Engineering) [6] is a methodology for web application development with ArgoUWE [5] as



1. selection list of topics obtained from topics!
2. event: View Messages
3. event: Post Message
4. event: Exit

Figure 4. Layout for Display Topics Event

a supporting CASE tool. One of the motivations for UWE is to provide support for complex business processes. Let us explore how we can apply the concept of atomic use case to UWE.

Example The example of an e-shop given in [5]. This example illustrates the basic steps and the basic models of UWE. The example shows four basic UWE models: the conceptual model, the navigation model, the (business) process structure model, and the process flow model. Part of the navigation model for the e-shop example is reproduced in Figure 5. It shows the navigation nodes, process nodes, navigation links (between navigation nodes), and process links (between navigation nodes and process nodes). One of the process nodes is the Checkout node, whose details are shown in the process flow model in Figure 6.

Analysis of the Checkout Process From the atomic use case point of view, we would perceive this Checkout node as consisting of two "aspects" which are associated with two different kinds of concerns: the first is about an atomic use case and the second is about the user interface design to support this atomic use case.

The atomic use case is the one that takes details about an order submitted by the user (such as customer name, address, credit card number, whether they want the items wrapped, etc.) and accordingly updates the underlying information base of the e-shop. We can call this the "Take Order" atomic use case (or "Place Order" from the customer's viewpoint).

With the Take Order atomic use case in mind, we can see that the essential role of the Checkout process model in Figure 6 (possibly apart from the "send invoice" action) is to obtain the details needed as input for the Take Order atomic use case. This is the second aspect we mentioned above. Viewing it this way, a question arises: How should we model this second aspect?

Instead of the process model in Figure 6, we could model the second aspect by a graphical user interface. For example, we can have a screen with two screen "blocks" that the user can interact with simultaneously:

– One block to confirm the items in the shopping cart and to select the wrapping options and

– One block to set the payment method.

The screen also has buttons to cancel or to place order. Such a screen would allow us to get the information we need, but it does not constraint us to the flow pattern described in the Checkout process flow in Figure 6. The process flow model is an over-specification: It is one way to get information to support the atomic use case but it is not the only one.

Possible Modifications to the UWE Models For the above Checkout process, it could be argued that the crucial point is the content of the information, rather how we obtain it. That is, the actual process in this case seems to play a secondary role.

For that reason, we could introduce to the UWE navigation model a new type of process node (through the stereotype mechanism) to represent processes with characteristics similar to those of the Checkout process above. We may call them "Use Case Process Nodes". A use case process node can be modeled by

– An activity diagram with a branching fork which has a number of alternatives: one of which leads to a "cancel", and each of the rest leads to an atomic use case (which represents the point where we update the information base)

– The specifications of the atomic use cases (we normally would have them already)

– A screen (which can be taken to be a part of presentation model) to indicate how the user can interact with the system to effect the atomic use case.

The suggested specification appears to be much simpler than what we currently have in UWE, and it avoids the problem of over-specification (which in most cases contains an element of arbitrariness on the part of the modeler).

By introducing the Use Case Process Node and by modeling it the way suggested above, we may bring about several advantages. First, there may be many instances of such processing nodes in an application, and it is clearer conceptually to single them out (to distinguish them from other kinds of processing nodes). Second, having singled them out, we can model them in a standard way as suggested above, which would simplify the specifying process.

Incidentally, another advantage is brought about by the act of modeling the atomic use cases itself. With current

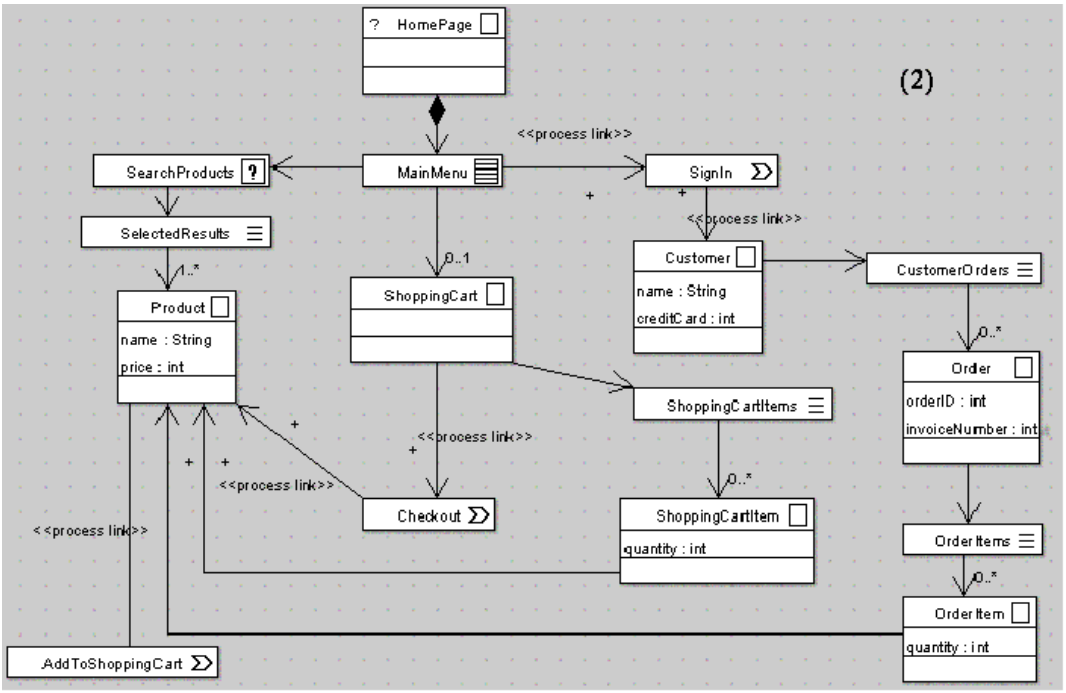


Figure 5. Navigation Model for the e-shop. Source: Knapp et al. [5]

UML modeling tools, the atomic use cases can be specified (more or less completely) with the help of OCL, and code (e.g. Java) can be generated out of the model. This generated code can quickly be enhanced to be a working prototype. This feature could be very attractive as a means to induce the average software engineer to apply OCL for rigorous modeling, which is an essential part of the MDE approach.

The admission of atomic use cases into the UWE model may open up a way to classify the typical business processes into various kinds of different grains of granuality. Some would be the use case process nodes described above. Some may represent external processes, which may be modeled by specifying the inputs and outputs. Still some may involve interactions with the user or external sources, which would require many of the UWE features for modeling.

8. Atomic Use Cases and the Travel Agency Case Study

Consider the Travel Agency Case Study described in [9]. We will give a sketch as to how one may approach this case study using atomic use cases. We will view the application as made up of the following types of components: the Personal Agent Assistant, the Broker Agent, the Transportation Company and the Finance Company.

The Personal Agent Assistant (or Assistant for short) is responsible for processing requests from the customers. To process a request, the Assistant maintains a list of Broker Agents, a list of Financial Companies, and data about the customer bookings and payment details. It would also need to store the current request and the list of current offers. We can think of the Assistant as a finite-state machine whose actions, which take place when transitions occur (some of which may be triggered by time constraints), would be the following atomic use cases:

– Enter Request (to store a user's request). Input: A request; Output: None; Pre: The request is valid; Post: Store the request.

– Get Offers (to send messages to Broker Agents to request offers) Input: The request data; Output: A list of offers; Pre: None; Post: Store the list of offers. (Note that we cannot impose the condition that the offers match the requirements of the request, though we do expect that to be the case.)

– Present Offers (to take the offers from the previous use case and to make the valid offers available to the user). Input: The set of offers (denoted by "InOffers"); Output: A set of offers (denoted by "OutOffers"); Pre: The request id exists; Post: OutOffers = the set of offers in InOffers that

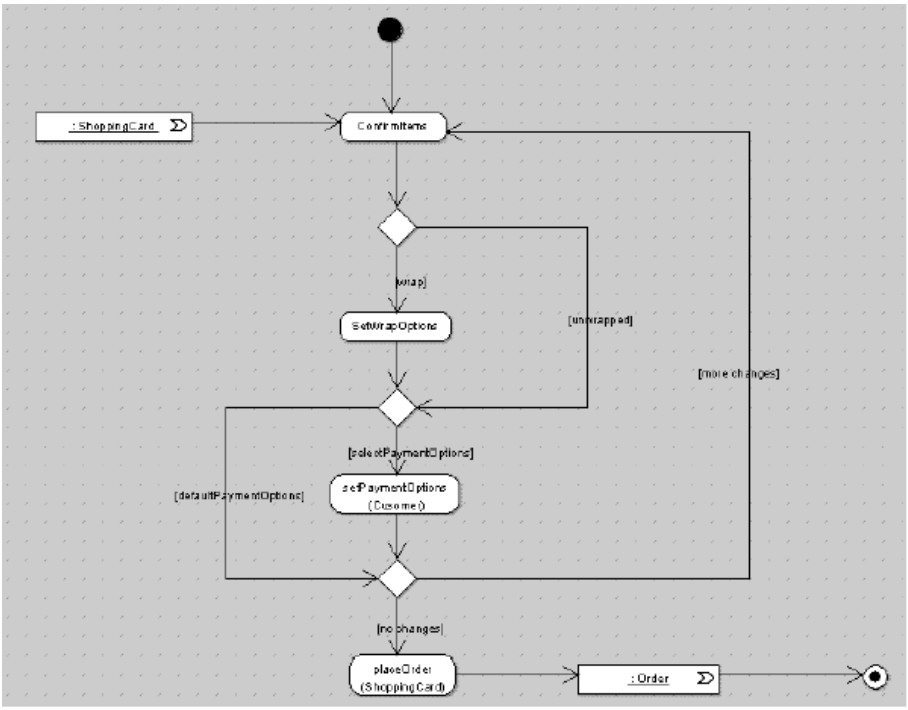


Figure 6. The Checkout Process Model. Source: Knapp et al. [5]

match the request.

– Make Booking (to take a booking (or bookings) on behalf of the user). Input: The offer identification, payment details; Output: Booking confirmation messages; Pre: The selected offer is one of the valid offers; Post: Validate payment details with Financial Company AND send messages to confirm the bookings for the selected offer and cancel the rest.

– Cancel Request (to cancel the request). Input: None; Output: None; Pre: None; Post: Delete current request and and cancel all offers

The Broker Agent would maintain a list of Transportation Companies. The main atomic use case it has are:

– Provide Offers (to take a request from the Assistant and to respond with a list of offer). Input: A request; Output: A list of offers; Pre: None; Post: The offers match the request AND store the offers.

– Confirm Booking (to confirm the booking of an offer which were previously temporarily booked). Input: The selected offer identification; Output: confirmation messages; Pre: The selected offer is in the current list of offers; Post: Send message to confirm bookings for selected offer and cancel the rest.

For the Transportation Company and the Finance Company, essentially we need to define the interface and the assumptions about the contracts related to the messages.

Having identified those atomic use cases, we can proceed to specify them formally, and construct the conceptual model (perhaps in the process of formalizing the atomic use cases), the navigation model, and the business process model.

9. Conclusion

In this paper, we have introduced the concept of atomic use case. It is a natural concept and therefore easy to grasp. In fact, it has appeared in various guises in the systems development literature. However, to really benefit from it, we need to have a clear understanding of the concept. Toward this end, we provided a definition (one that aims to assist us in identifying the atomic use cases). We showed how we can identify the atomic use cases and specify them, and how we can build a complete business layer with them. Finally, we explored how we can incorporate the concept into UWE for web application development, which could lead to a more definite choice of granularity for the process nodes and clearer relationships between the various models.

References

[1] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Comput. Sci.* , 285:187-243, Sept. 1995.

[2] Alistaire Cockburn *Writing Effective Use Cases*, Addison-Wesley, 2001.

[3] R. Duke and G. Rose (2000) *Formal Object-Oriented Specification Using Object-Z*, MacMillan.

[4] Jacobson J., Ericsson M. and Jacobson P. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*, Wokingham:Addison-Wesley.

[5] Alexander Knapp, Nora Koch, Gefei Zhang, and Hanns-Martin Hassler. Modeling Business Processes in Web Applications with ArgoUWE, In *7th International Conference on the Unified Modeling Language (UML2004)*, LNCS 3273, 69-83, Springer Verlag, October 2004.

[6] Nora Koch and Andreas Kraus. The expressive Power of UML-based Web Engineering. In *Second International Workshop on Web-oriented Software Technology (IWWOST02)*, D. Schwabe, O. Pastor, G. Rossi, and L. Olsina, editors, CYTED, 105-119, June 2002.

[7] Kinh Nguyen, Tharam Dillon, Atomic Use Case: A Concept for Precise Modelling of Object-Oriented Information Systems, in *OOIS'03, The Ninth International Conference on Object-Oriented Information Systems*, Geneva, Switzerland, 2003

[8] Kinh Nguyen. *A Semi-Formal Object-Oriented Method for Analysis and Modelling of the Functional Requirements of Information Systems*, PhD Thesis in Computer Science, La Trobe University, 2003.

[9] A Travel Agency System, Case Study for Workshop on model-driven Web Engineering (MDWE 2005), http://www.lcc.uma.es/ av/mdwe2005/TheTAEexample